AIAA-98-4531

A METHOD TO INTERFACE AUTO-GENERATED CODE INTO AN OBJECT-ORIENTED SIMULATION FRAMEWORK

Patricia C. Glaab, Kevin Cunningham* , P. Sean Kenney,
Richard A. Leslie, David W. Geyer, Michael M. Madden*

Unisys Corporation
NASA Langley Research Center
Mail Stop 125B
Hampton, VA 23681

Abstract

Sophisticated computer-based tool packages allow control system researchers to develop and analyze complex control systems from their desktops. Tools such as Matrix X and Matlab are capable of producing auto-generated code of control system diagrams as an output option. Validation requirements are greatly reduced when the auto-generated code can be directly installed into an engineering simulation program.

The code produced by automatic code generators, however, is often cryptic and inflexible. Incorporating it elegantly into an object-oriented simulation requires forethought on the part of the simulation programmer. This paper presents a method of interfacing auto-generated code into a flexible, object-oriented simulation framework that maintains encapsulation of objects within the framework and does not compromise the design of the system.

Introduction

Support for flexible software design methods is a goal of object-oriented simulation program development at the NASA Langley Research Center (LaRC). The ability to accommodate auto-generated code from sophisticated computer-based tool packages provides a versatile analysis environment for researchers and potentially reduces the workload of the simulation programmer. The auto-generated code module, however, has very little flexibility when it must be ported directly from the tool output to the simulation environment. A technique was required to interface between auto-coded modules and the simulation framework that did not compromise the object-oriented design of the simulation program.

The methodology presented here was developed to support the baseline simulation of a commercial transport autopilot/flight director system (AFDS) developed within the Langley Standard Real-time Simulation in C++ (LaSRS++) [2] at LaRC. The AFDS implementation in the simulation was structured to allow auto-generated code to be easily incorporated into the C++ simulation program. Specific requirements for this design included two major objectives. The final design required rapid linking of auto-coded modules into a soft real-time simulation without having to modify simulation or auto-generated code. This was intended to minimize the potential of accidentally changing the code's behavior from its original configuration. Addi-

* Senior Member, AIAA

tionally, hand-coded versions of all AFDS subsystem objects were required as part of the baseline, and either of the two versions of each component must be selectable between runs. For the auto-coded versions, the intention was for a smooth and logical transition from a desktop tool environment to a soft real-time environment, and eventually to a hard real-time session with a cockpit and pilots. Other simulation users required the unmodified AFDS at a much earlier date for support of other studies, and since not all subsystem components would necessarily be delivered as auto-code, a complete hand-coded version was required as the default. The method developed allows the simulation user to select the version of the AFDS, baseline C++ or linked auto-code, for each component of the 757 autopilot independently.

The intention of this paper is to present a design method for managing complex, interacting systems with a sound overall object-oriented (OO) policy that ultimately allows flexibility at the lowest object levels. By designing a system architecture that meets these goals, incorporating code with special input/output (I/O) requirements is simplified at startup and for future maintenance. Auto-generated code created by computer-based engineering tool presents special I/O requirements because the tool manufacturer chooses the style of the interface and execution. These conventions must be gracefully accommodated if the code is to be ported to the simulation environment unaltered.

Like many legacy systems that were written before the advent of OO design philosophy, the prior implementation of the system relied heavily on component interaction and data sharing in a web-like fashion. The task of implementing an OO system heirarchy within the target system as a precursor to interfacing auto-code was included because this problem presents itself for many complex, legacy systems. It is also crucial to the clean incorporation of the auto-generated code. Whenever possible, established design patterns were used in the system architecture. Design patterns offer simple and elegant solutions to specific problems that tend to occur over and over in software problem solving. By incorporating time-tested solutions, a programmer may capitalize on the experience and efforts of many developers and begin a design at a stage that may have otherwise required years of refinement [1].

An Overview of the Example System

The documentation received for the simulated AFDS defines the system as a complex interaction of many components. Each of the components has unique I/O requirements, and all are highly dependent on each other. Though these component definitions provide a logical delineation for class objects, their interdependence violates encapsulation and would make unit testing of individual subsystem components difficult.

A hand-coded C++ implementation was required to function as part of the simulation framework to emulate standard behavior of the AFDS. This version was developed using the same style guide conventions imposed on the overall simulation program. The second version was an auto-generated code module delivered by researcher engineers using a computer-based engineering tool package. The behavior of this version would change as research required. The complexity of accommodating two different I/O format requirements for many components portended a maintenance nightmare. Also, the delivery of each auto-coded module was uncertain. . The absence of any or all auto-code modules could not break the system. Rapid reconfiguration and testing was required as the auto-coded components were delivered or modified. Three design criteria were established to meet final code requirements:

- The web-like interaction of the components as presented in the original documentation had to be removed and the I/O managed more cleanly
- An isolated location had to be created to perform the specific I/O requirements for the auto-code without propagating knowledge or burden of these special requirements to other parts of the system
- A foundation had to be laid to effectively isolate the internal complexity of component objects from the client code that would use them

Step One: Untangling the Web

An ideal unit test scenario allows the developer to begin testing at the lowest level in an isolated development area without having to depend on other objects being linked in or emulated. The object should then be portable to the production environment without modification. With a web of objects that directly interact with each other, unit testing becomes extremely difficult. References to data sources included in a production version of a class must be removed, or the referenced sources must be included in the test environment. As links within the class grow, so does the size and complexity of the unit test environment.

Though loose coupling of objects is common in modern OO code designs, isolation of code modules in legacy system may not be so clearly defined. The quick and easy inclusion of common blocks in FORTRAN coding, for example, is conducive to a final design that uses a web of data interaction.

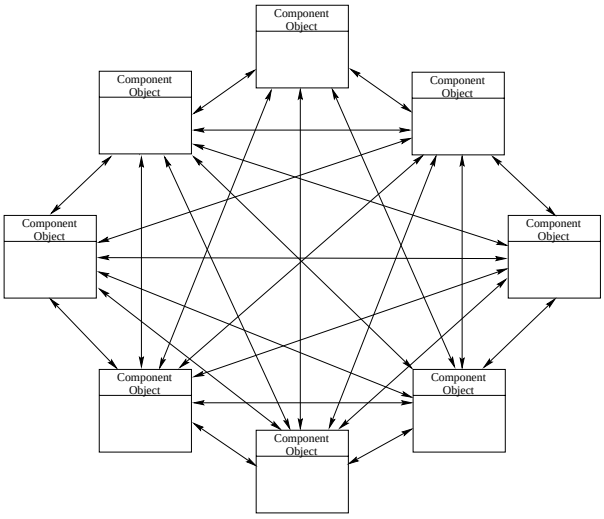


Figure 1 - Web of interaction when object are allowed to refer directly to each other

The design of the example AFDS system was developed in a procedural FORTRAN environment and relied heavily on information sharing and passing between objects. As a first step in untangling the interactive web, a Mediator class [1] was created to manage information to the subsystem components. The Mediator class encapsulates how the objects within the system may interact and orchestrates the exchange of inputs and outputs between subsystem components. This code pattern promotes loose coupling by keeping objects from referring to each other directly, and it lets the developer vary their behavior independently. By imposing this Mediator to control interaction, the tight coupling shown in the previous illustration was removed.

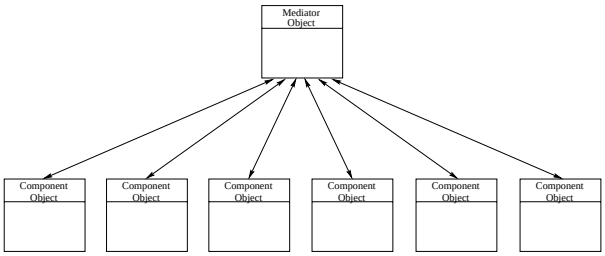


Figure 2 - Controlled interaction using a Mediator class

Each component object receives inputs from and provides outputs to the Mediator class exclusively. No component within the system has knowledge or dependence on any other component. This loosely coupled system removes interdependencies of subsystems and provides a clean, manageable point of reference for all subsystem I/O.

Step 2: Isolating and Adapting the Special I/O Requirements of the Auto-Generated Code

With auto-generated code delivered by an outside source, the simulation developer may not have control over the style of data input or function execution. Chances are slim that the manufacturer of the computer-based tool package chose the same style guide conventions adopted by the target simulation program. When several versions of a subsystem component are required, this necessitates several styles of communication to handle data transfer and operation. If the auto-code version is not stable (subject to frequent changes in a research environment, for example), the changing communication requirements translate to maintenance overhead that can propagate through the entire subsystem. This maintenance overhead can be isolated within one class with the use of an Adapter pattern [1].

The Adapter class converts the interface of a class into another interface that the client expects. This allows the

client to communicate with the auto-code in a fashion consistent with local style guide conventions. A standard method of invoking execution of the auto-code can also be defined that need not be changed if the delivered auto-code changes significantly. The Adapter class can simply redefine the action upon the auto-code internal to itself.

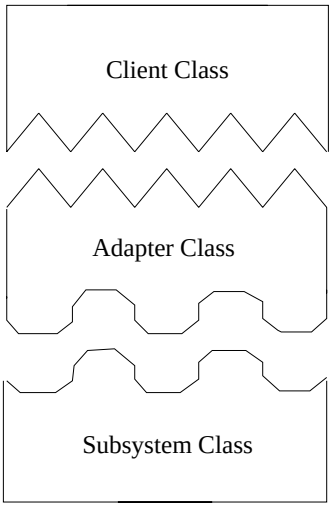


Figure 3 - Adapter class addition to translate interface

Adding the Adapter class layer becomes especially beneficial when the information sent to the various component versions is generally the same. In this case, the interface can be defined in one base class from which all components interfaces inherit. This isolated communication point provides an ideal place to set up a Facade.

Step 3: Hiding the Complexity of Component Behaviors and Interfaces

The decision requirements to accommodate two or more interface versions for each subclass component present a significant increase in the complexity of the client class (the Mediator class in the example system). Additionally, maintenance work may be required if interface requirements change. To remove knowledge of the complex interactions of the subsystems from the classes that use them, a unified interface can be defined to make the subsystem to make easier to use. This unified interface is called a Facade [1] and was used in the example system to encapsulate decision-making overhead within the operations of the subclass components themselves. In the C++ implementation of the AFDS component, the Facade is the base class from which the hand-coded implementation inherits. It is also the base class from which the C++ Adapter class to the auto-generated code inherits.

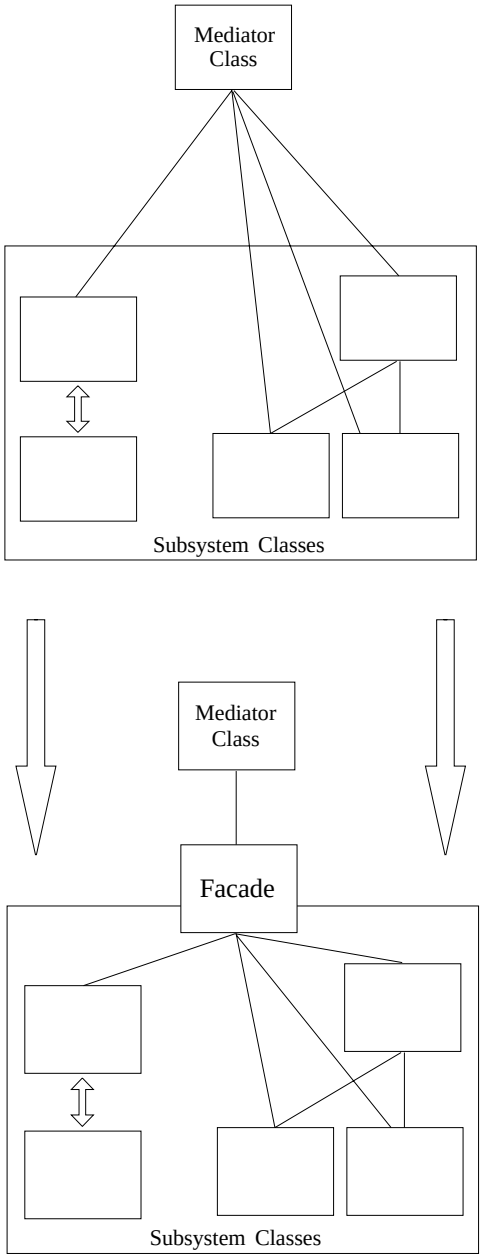


Figure 4 - Facade addition to reduce apparent complexity to client

In a simpler scenario where only one version of a subsystem must be interfaced, the Facade is not necessarily needed as part of the design. This is true as long as the client class, the Mediator, is only required to communicate to an interface isolated within one subsystem class and that interface is stable. For example, in this implementation with only auto-generated code as the subsystem target, the Mediator would speak directly to the Adapter class.

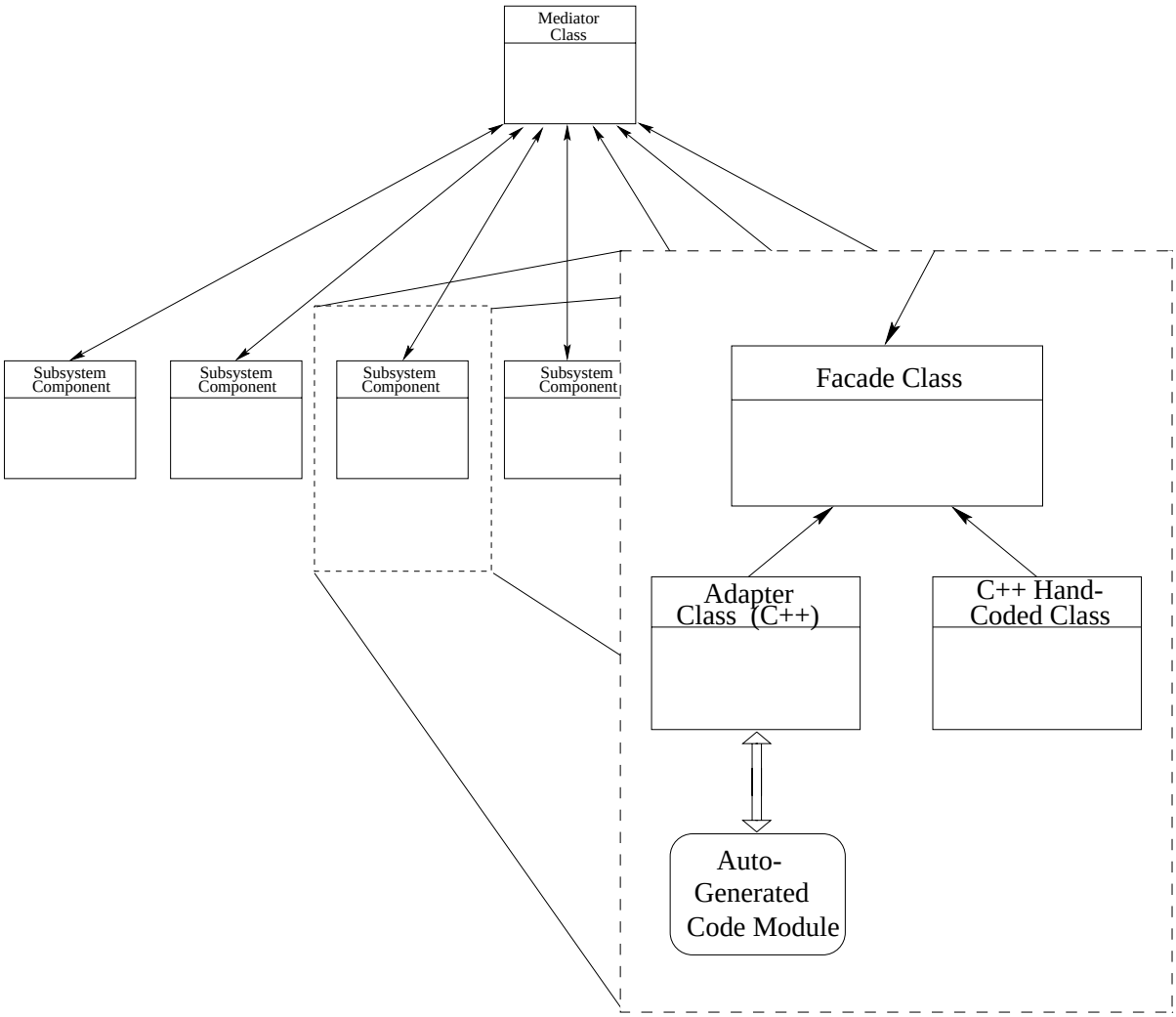
Unit Testing

Unit testing is handled easily at several levels in the final system architecture. The auto-generated version of the system is an isolated entity and its inputs and outputs and internal functionality are easily exercised in a stand-alone fashion. The hand-coded subsystem object uses an isolated, well-defined interface and a unit test program can either be imposed at the level of the Facade for exercising the client only, or above the Facade to exercise the Facade interface to the client code.

At a higher level, the two implementations can be tested or compared from a single unit test program which takes advantage of the one unified Facade interface.

Final Design and conclusions

When inflexible interfaces must be accommodated into an OO environment, a clean initial design of the system is critical to isolating the external component from the



simulation framework. Isolation of the interface then minimizes compromises that must be make to style guide conventions. By using the techniques outlined in this paper, an AFDS was implemented in the LaSRS++ framework that met the required design criteria. All code specialized to interface to auto-generated code modules is encapsulated into one class per component and hidden from the clients that use it. Code maintenance to support auto-code changes are localized within this one Adapter class. Selection of component versions is provided to the simulation user, yet facilitated to the information handling class (the Mediator) through the use of a Facade software pattern that provides one unified interface to all subsystems. The example system describes implementation of Xmath/SystemBuild auto_code in C, but theoretically the technique could be used to interface any type of code with special I/O requirements. A related approach is taken in the hardware interface of the LaSRS++ simulation with cockpit controls, displays, flight management computers, etc. as a method of isolating hardware-specific code requirements through abstraction [6].

Bibliography

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[2] Richard A. Leslie, et al. LaSRS++ An Object-Oriented Framework for Real-Time Simulation of Aircraft. Paper Number AIAA-98-4529, August, 1998.

[3] Keith D. Hoffler, Dr. Thomas E. Alberts, and Patricia C. Glaab. Implementing the Control Laws for the NASA Transport Systems Research Vehicle B-757. ViGYAN, Inc., Hampton, Virginia, 1997.

[4] Steve McConnell. Code Complete: A Practical Handbook of Software Construction. Microsoft Press, Redmond, Washington, 1993.

[5] Scott Meyers. Effective C++. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[6] P. Sean Kenney, et al. Using Abstraction To Isolate Hardware In An Object-Oriented Simulation. Paper Number AIAA-98-4533, August, 1998.

[7] Grady Booch. Object-Oriented Analysis and Design. Benjamin/Cummings, Redwood City, California, 1994.